Chapter 6. Custom Serialization

You may find that your web services and client applications work perfectly well using the set of data types supported by your SOAP implementation. After all, the technologies we're looking at provide a great deal of flexibility with their support for arrays and custom types. On the other hand, somewhere along the line you may need to serialize something that simply isn't possible using the techniques we've covered up to this point. Maybe you want to serialize the data for a Java class in a way that is more consistent with your existing applications, even if the encoding is not industry standard. Or maybe you want to add support for constructs that are described in the SOAP specification but are not supported in the available SOAP implementations.

The implementation of SOAP that you use may force you to write custom serializers for certain types of objects. For example, Apache provides the Bean serializer; it could provide other special case serializers, and it's not required to provide that one. So the conditions under which you need to supply a custom serializer are implementation dependent.

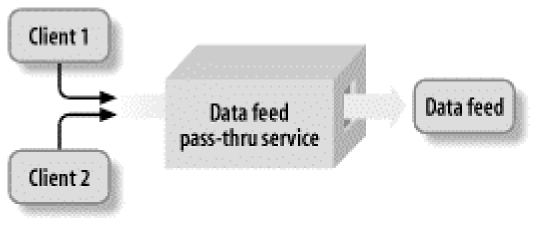
It's also important to consider the usefulness of the data being serialized. Many of Java's rules about what implements the `Serializable` interface make sense here as well. For example, SOAP or not, serializing an I/O stream is not likely a sensible thing to do. This is true for any kind of distributed system. You have to understand the semantics of the data before blindly sending it to another party.

We'll tackle a few examples of custom serialization in this chapter, exploring the techniques and APIs available in both Apache SOAP and GLUE. Unlike in previous chapters, we won't repeat the examples using both technologies. By now, you should be sufficiently familiar with both Apache SOAP and GLUE to have an idea of how to work with each.

6.1 Custom Type Encoding

Let's start off by working with a custom data type that uses a nonstandard serialization. We'll develop this example using Apache SOAP. Imagine that we're developing a web service that acts as a pass-thru proxy to a proprietary stock market data feed. We want to send the data to the service already serialized in the form used by the feed. Yep, another contrived example! But by the time we finish it, I bet you'll have thought of a few uses for this technique. The service is designed to allow client applications to pass agreed-upon formats to the downstream data feed service without having to modify the web service itself. So the serialized data format is not known to the web service; it's understood only by the client application and the data feed. Figure 6-1 shows what this architecture might look like. Both of the client applications use the same service and invoke the same methods. However, they may or may not be using the same serialization format for the data to be sent to the data feed.

Figure 6-1. A pass-thru service



Let's say that the default data format for the feed is a linear self-describing format. This format is an arbitrary-length string that contains a variable number of named data fields and their string values. The format can accept from 1 to 9 fields. The first character of the message contains the number of fields, and can be any ASCII character between 1 and 9. Each field consists of the name of the field followed by a colon, then the field value, followed by another colon. Here's an example of a message that contains two fields named SYMBOL and PRICE, with corresponding values of ZZ and 110.5:

```
2 S Y M B O L : Z Z : P R I C E : 1 1 0 . 5 :
```

This is the format we want to use to serialize messages. But first, let's create a new service called `urn:DataFeedService` that contains a method called `sendMessage`. This method takes a single message parameter and returns a message parameter. So before we begin to define the Java class for the service, we need to define the Java class for the message. The `DataFeedMessage` class represents a message to be used for this service. It contains a list of name/value pairs for keeping track of the field names and their data values.

```
package javasoap.book.ch6;
import java.util.*;
public class DataFeedMessage {
   Hashtable _fields = new Hashtable( );
   public DataFeedMessage( ) {
   }
   public void addField(String name, String value) {
      _fields.put(name, value);
   }
   public Enumeration getFieldNames( ) {
      return _fields.keys( );
   }
   public String getFieldValue(String name) {
      return (String)_fields.get(name);
   }
   public String toString( ) {
      String cnt = String.valueOf(_fields.size( ));
      StringBuffer msg = new StringBuffer( );
      msg.append(cnt);
      msg.append(":");
      for (Enumeration e =_fields.keys(); e.hasMoreElements( );) {
         String name = (String)e.nextElement( );
         String value = (String)_fields.get(name);
         String section = name + ":" + value + ":";
         msg.append(section);
      }
      return msg.toString( );
   }
   public void parseFormattedMessage(String msg) {
      _fields.clear( );
      StringTokenizer st = new StringTokenizer(msg, ":");
      String token = st.nextToken( );
      int cnt = new Integer(token).intValue( );
      for (int i = 0; i < cnt; i++) {
         String name = st.nextToken( );
         String value = st.nextToken( );
         addField(name, value);
      }
   }
```

The fields are stored in the _fields variable, which is an instance of java.util.Hashtable. The addField( ) method allows you to add a field by providing its name and value as parameters, which are then added to the hash table using the name as the key. The getFieldNames( ) and getFieldValue( ) methods provide access to the hash table element names and values without exposing the fact that the class uses a java.util.Hashtable internally. The last two methods are the most interesting. The toString( ) method returns a string containing all of the fields and their data values in the message format we defined earlier. The parseFormattedMessage( ) method takes a string parameter in the described format and populates the internal hash table with the contents. These two methods will be helpful when we want to serialize and deserialize our custom message type.

Now we can create the Java class for this new service. We'll call the class DataFeedService, and we'll include a method called sendMessage( ) that takes an instance of DataFeedMessage as its parameter. This method returns an instance of DataFeedMessage as well. Here is the DataFeedService class:

```
package javasoap.book.ch6;
public class DataFeedService {
   public DataFeedService( ) {
   }
   public DataFeedMessage sendMessage(DataFeedMessage msg) {
      // we could pull the formatted message here
      // by calling msg.toString( );
      String stock = msg.getFieldValue("SYMBOL");
      String request = msg.getFieldValue("REQUEST");
      DataFeedMessage res = new DataFeedMessage( );
      res.addField("SYMBOL", stock);
      if (request.equals("PRICE"))
         res.addField("PRICE", "123.5");
      return res;
   }
}
```

All we're doing in the example is grabbing the stock symbol from the data feed message, using it as the stock symbol for the return value, and adding a price field to the return value as if the field named REQUEST has a value of PRICE. So the sendMessage( ) method uses some of the data from the input parameter to create a return value of type DataFeedMessage.

Because this class doesn't obey the JavaBeans method naming patterns, we can't use the BeanSerializer provided by Apache SOAP to serialize and deserialize instances of the data feed message we've created. So we'll create a serializer of our own. A custom Java class for serializing objects must implement the org.apache.soap.util.xml.Serializer interface; to implement this interface, we must write a marshall( ) method. Similarly, a Java class for deserializing objects must implement the org.apache.soap.util.xml.Deserializer interface and provide the unmarshall( ) method. It's perfectly reasonable to create one class that handles both functions, so we'll take that approach. Let's call our custom serializer class DataFeedMessageSerializer .

```
package javasoap.book.ch6;
import org.apache.soap.encoding.soapenc.*;
import org.apache.soap.util.xml.*;
import org.apache.soap.util.*;
import org.apache.soap.rpc.SOAPContext;
import org.w3c.dom.*;
import java.io.*;
import java.util.*;
public class DataFeedMessageSerializer
                 implements Serializer, Deserializer {

  public DataFeedMessageSerializer(  ) {
  }
  public void marshall(String inScopeEncStyle, Class javaType,
                Object src, Object context, Writer sink,
                NSStack nsStack, XMLJavaMappingRegistry xjmr,
                SOAPContext ctx)
        throws IllegalArgumentException, IOException  {

    if(!javaType.equals(DataFeedMessage.class)) {
      throw new IllegalArgumentException(
        "Can only serialize javasoap.book.ch6.DataFeedMessage instances");
    }

    nsStack.pushScope(  );

    if (src != null)
    {
      SoapEncUtils.generateStructureHeader(inScopeEncStyle,
                                        javaType, context,
                                        sink, nsStack, xjmr);
      DataFeedMessage msg = (DataFeedMessage)src;
      String data = msg.toString(  );
      sink.write(data);
      sink.write("</" + context + '>');
    }
    else
    {
      SoapEncUtils.generateNullStructure(inScopeEncStyle,
                                        javaType, context,
                                        sink, nsStack, xjmr);
    }
    nsStack.popScope(  );
  }
  public Bean unmarshall(String inScopeEncStyle, QName
                elementType, Node src,
                XMLJavaMappingRegistry xjmr, SOAPContext ctx)
          throws IllegalArgumentException {

    Element elem = (Element)src;
    String value = DOMUtils.getChildCharacterData(elem);

    DataFeedMessage msg = null;
    if(value!=null && !((value=value.trim(  )).equals("")) {
       msg = new DataFeedMessage(  );
       msg.parseFormattedMessage(value);
    }
    return new Bean(DataFeedMessage.class, msg);
  }
}
```

Let's look first at the `marshall( )` method, which serializes an instance of `DataFeedMessage`. It first verifies that the `javaType` parameter value is `DataFeedMessage.class`, because this custom serializer can't serialize an instance of any other object. If the class is wrong, `marshall( )` throws an `IllegalArgumentException` with an appropriate description. (We haven't discussed exceptions yet, so you may be wondering what the result of throwing one would be. Hold that thought until Chapter 7, which delves into that subject.) Next, we call `nsStack.pushScope( )`. The `nsStack` variable that was passed as a parameter represents the hierarchical namespace of the SOAP message being serialized. This object serves as a stack of namespace definitions. Each time the `marshall( )` method of a serializer is called, we push the namespace scope. This technique allows us to make namespace declarations at our level of the hierarchy. Before returning from the `marshall( )` method, we must be sure to pop the stack. If we don't, we run the risk of overriding the namespace declarations of the XML elements above us in the hierarchy.

Now that we've pushed the namespace scope, we make sure that the object that we're supposed to serialize isn't null. If `src` is null, we call the `SoapEncUtils.generateNullStructure( )` method, which handles the creation of a null element for us. If `src` is not null (the more interesting case), we call `SoapEncUtils.generateStructureHeader( )` as the first step in creating a SOAP message. This method creates the element tag, and writes any attributes and associated values that are needed. We don't have to handle any of that; our job is to deal with the serialization of the actual data, not the XML tag details. Well, that's almost the truth. We do some cleanup work at the end, as you'll see shortly.

Since we know that `src` is the right type, we cast it to `DataFeedMessage` and save a reference to it in the `msg` variable. All we have to do now is call the `msg.toString( )` method, which returns the data of the data feed message in the required format. The properly formatted message is now stored in the `data` variable. The `marshall( )` method takes a parameter called `sink`, which is an instance of `java.io.Writer`. This is where all of the serialization for the SOAP envelope takes place. At this point, `sink` is positioned at the point where the actual serialized data belongs. Calling the `sink.write( )` method using `data` as the parameter writes the formatted data feed message to the `sink`. The last step is to write the terminating tag for this element, even though writing the element tag was handled for us. We'll need the name of the element to write its closing tag; that's available in the `context` object passed as one of the parameters to `marshall( )`. We call `sink.write( )` one more time to close the element. This works because `context` implements the `toString( )` method, which returns the name of the element being serialized. That's all there is to it. Pop the namespace stack by calling `nsStack.popStack( )`, and we're done.

Now let's look at the process of deserializing: turning the encoded data into an instance of a Java class. This process is handled by the `unmarshall( )` method of the serializer. Deserialization requires the use of some XML processing APIs provided by Apache SOAP. We cast the `src` parameter into an instance of `org.w3c.dom.Element`, which is a class used to represent an XML element. Next, we want to get at the formatted data of the element. That's the data feed message in the format described at the start of this section. This is accomplished by passing the `elem` variable to the `getChildCharacterData( )` method of the `org.apache.soap.util.xml.DOMUtils` class. The string value returned from that call, saved in `value`, contains the formatted data feed message. Of course, it's a good idea to make sure there is actually data in the message, which is why we check for a null or empty string first. If there is data, a new instance of `DataFeedMessage` is created and the data is passed to its `parseFormattedMessage( )` method. At this point, we've successfully deserialized the data and used it to populate an instance of `DataFeedMessage`. The `unmarshall( )` method needs to return an instance of `org.apache.soap.util.Bean`;[1] this class is used to associate a Java object with its Java class. You may be wondering why this is necessary when you can determine the class of an object by calling its `getClass( )` method. However, calling `getClass( )` is problematic if the object is implemented using a subclass of the mapped class, and it is impossible if the object is `null`. So the last step is creating an instance of `org.apache.soap.util.Bean`, using `DataFeedMessage.class` as the first constructor parameter and `msg` as the second.

We've now implemented the serialization and deserialization capabilities of our custom serializer. Now we can go ahead and deploy `urn:DataFeedService`. Deploying the service requires that we include an appropriate mapping entry in the deployment descriptor so that instances of our custom type are mapped to the correct Java class, and associated with the custom serializer we just created. Here's the deployment descriptor:

```
<isd:service
    xmlns:isd="http://xml.apache.org/xml-soap/deployment"
    id="urn:DataFeedService">
  <isd:provider
     type="java"
     scope="Application"
     methods="sendMessage">
    <isd:java
       class="javasoap.book.ch6.DataFeedService"
       static="false"/>
  </isd:provider>

  <isd:faultListener>org.apache.soap.server.DOMFaultListener
  </isd:faultListener>
  <isd:mappings>
    <isd:map
       encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
       xmlns:x="urn:DataFeedService" qname="x:DataFeedMessage"
       javaType="javasoap.book.ch6.DataFeedMessage"
         java2XMLClassName="javasoap.book.ch6.DataFeedMessageSerializer"
         xml2JavaClassName="javasoap.book.ch6.DataFeedMessageSerializer"/>
  </isd:mappings>
</isd:service>
```

The mapping section has an entry for `DataFeedMessage`, namespace-qualified using the name of the service itself. We assign the value `javasoap.book.ch6.DataFeedMessageSerializer` to both the `java2XMLClassName` and the `xml2JavaClassName` attributes, which tells the system about our custom serializer class. The service can now be deployed.

Here's a client application that accesses this service. The process is similar to the one we used in the last chapter: we map our custom types as beans and use the `DataFeedMessageSerializer` to serialize and deserialize the object.

[1] Don't be confused by the fact that this class is called a `Bean`. It's not specifically related to the `BeanSerializer` class we worked with in the previous chapter; it's just a name left over from some other project.

```
package javasoap.book.ch6;
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.encoding.*;
import org.apache.soap.util.xml.*;
public class DataFeedClient {
   public static void main(String[] args)
      throws Exception {

      URL url = new URL(
          "http://georgetown:8080/soap/servlet/rpcrouter");

      Call call = new Call( );

      SOAPMappingRegistry smr = new SOAPMappingRegistry( );
      call.setTargetObjectURI("urn:DataFeedService");
      call.setMethodName("sendMessage");
      call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
      call.setSOAPMappingRegistry(smr);
      DataFeedMessageSerializer msgSer =
                   new DataFeedMessageSerializer( );
      // Map the data feed message type
      smr.mapTypes(Constants.NS_URI_SOAP_ENC,
            new QName("urn:DataFeedService", "DataFeedMessage"),
            DataFeedMessage.class, msgSer, msgSer);
      DataFeedMessage msg = new DataFeedMessage( );
      msg.addField("SYMBOL", "XYZ");
      msg.addField("REQUEST", "PRICE");
      Vector params = new Vector( );
      params.addElement(new Parameter("msg",
                   DataFeedMessage.class, msg, null));
      call.setParams(params);

      try {
         Response resp = call.invoke(url, "");
         Parameter ret = resp.getReturnValue( );
         Object value = ret.getValue( );
         System.out.println(value);
      }
      catch (SOAPException e) {
         System.err.println("Caught SOAPException (" +
                         e.getFaultCode( ) + "): " +
                         e.getMessage( ));
      }
   }
}
```

We create an instance of `DataFeedMessageSerializer` in the variable called `msgSer`. When we call `smr.mapTypes( )` to map our custom type to the `DataFeedMessage` class, we pass `msgSer` as both the serializer and the deserializer. So when we invoke the `sendMessage` service method, the `DataFeedMessageSerializer` is used to serialize the data feed message, and also to deserialize the return value from the service method. An instance of `DataFeedMessage` is created, and fields are added named SYMBOL and REQUEST, with data values of XYZ and PRICE, respectively. Now we set up the service method parameters and invoke the method. The return value from the `sendMessage` service method is also a data feed message, so it is deserialized into an instance of `DataFeedMessage`. We don't bother to cast the return of `ret.getValue( )`, because we can take advantage of the `toString( )`

method to display its contents. When you run this example you should get the following output:

```
2:SYMBOL:XYZ:PRICE:123.5:
```

This is the formatted data from the returned instance of `DataFeedMessage`; the server put the symbol that was passed to it into the return value, along with a price field with a value of 123.5. Let's take a look at the SOAP envelope that was sent to the server:

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <SOAP-ENV:Body>
     <ns1:sendMessage
        xmlns:ns1="urn:DataFeedService"
        SOAP-ENV:encodingStyle=
          "http://schemas.xmlsoap.org/soap/encoding/">
       <msg
       xsi:type="ns1:DataFeedMessage">2:REQUEST:PRICE:SYMBOL:XYZ:
       </msg>
     </ns1:sendMessage>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The `msg` element is typed as `DataFeedMessage`, namespace-qualified with `urn:DataFeedService`. You can see that the data is formatted according to our specification. Now let's look at the SOAP envelope returned by the service method, which also types the returned data as a `DataFeedMessage` using the `urn:DataFeedService` namespace. So we've been able to use our custom serialization classes to handle the marshalling and unmarshalling of our data feed message on both the server and the client.

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <SOAP-ENV:Body>
     <ns1:sendMessageResponse
       xmlns:ns1="urn:DataFeedService"
       SOAP-ENV:encodingStyle=
         "http://schemas.xmlsoap.org/soap/encoding/">
         <return
         xsi:type="ns1:DataFeedMessage">2:PRICE:123.5:SYMBOL:XYZ:
         </return>
    </ns1:sendMessageResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

6.1.1 Serializing Sparse Arrays

We discussed sparse arrays back in Chapter 3. However, neither Apache SOAP nor GLUE supports them. So let's do it ourselves.

Writing a sparse array serializer from scratch could be a lot of work, so we start by grabbing the source code for `org.apache.soap.encoding.soapenc.ArraySerializer` from the Apache SOAP distribution. We'll modify the `ArraySerializer` code to create a new class called `javasoap.book.ch6.SparseArraySerializer`. Here's the code for the entire class, but don't be overwhelmed. Most of it came from the `ArraySerializer` class. We'll just walk through the modifications for handling sparse arrays.

```
package javasoap.book.ch6;
import org.apache.soap.encoding.soapenc.*;
import java.beans.*;
import java.io.*;
import java.util.*;
import java.lang.reflect.*;
import org.w3c.dom.*;
import org.apache.soap.util.*;
import org.apache.soap.util.xml.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
public class SparseArraySerializer implements Serializer, Deserializer
{
   public SparseArraySerializer(  ) {
   }
   public void marshall(String inScopeEncStyle,
            Class javaType, Object src,
            Object context, Writer sink, NSStack nsStack,
            XMLJavaMappingRegistry xjmr, SOAPContext ctx)
       throws IllegalArgumentException, IOException {
      nsStack.pushScope(  );
      String lengthStr = src != null
                    ? Array.getLength(src) + ""
                    : "";
      Class componentType = javaType.getComponentType(  );
      QName elementType = xjmr.queryElementType(componentType,
                               Constants.NS_URI_SOAP_ENC);
      if (src == null) {

         SoapEncUtils.generateNullArray(inScopeEncStyle,
                                javaType,
                                context,
                                sink,
                                nsStack,
                                xjmr,
                                elementType,
                                lengthStr);
      }
      else {

         SoapEncUtils.generateArrayHeader(inScopeEncStyle,
                                javaType,
                                context,
                                sink,
                                nsStack,
                                xjmr,
                                elementType,
                                lengthStr);
        sink.write(StringUtils.lineSeparator);
        int length = Array.getLength(src);
        for (int i = 0; i < length; i++) {

           nsStack.pushScope(  );
           Object value = Array.get(src, i);
        // we only want to serialize if the
        // element exists, since
        // this is a sparse array serializer
        if (value != null) {
         Class actualComponentType = value.getClass(  );
          // use a temporary sink so that we can
          // modify it before writing it out
          StringWriter sw = new StringWriter(  );
          xjmr.marshall(inScopeEncStyle,
                   actualComponentType, value, "item",
             sw, nsStack, ctx);

         sink.write("<item ");
          sink.write(nsStack.getPrefixFromURI(
              Constants.NS_URI_SOAP_ENC) +
             ":position=\"[" + i + "]\" ");
         sink.write(sw.toString(  ).substring(6));
         sink.write(StringUtils.lineSeparator);
       }
       nsStack.popScope(  );
     }
     sink.write("</" + context + '>');
   }
   nsStack.popScope(  );
}
public Bean unmarshall(String inScopeEncStyle,
                QName elementType, Node src,
                XMLJavaMappingRegistry xjmr, SOAPContext ctx)
           throws IllegalArgumentException {

   Element root = (Element)src;
   String name = root.getTagName(  );
   QName arrayItemType = new QName("", "");
   Object array = getNewArray(inScopeEncStyle, root,
                         arrayItemType, xjmr);
   if (SoapEncUtils.isNull(root)) {
      return new Bean(array.getClass(  ), null);
   }
   Element tempEl = DOMUtils.getFirstChildElement(root);
   while (tempEl != null) {
      String declEncStyle = DOMUtils.getAttributeNS(tempEl,
             Constants.NS_URI_SOAP_ENV,
             Constants.ATTR_ENCODING_STYLE);
      String actualEncStyle = declEncStyle != null
                         ? declEncStyle
                         : inScopeEncStyle;
      QName declItemType =
             SoapEncUtils.getAttributeValue(tempEl,
             Constants.NS_URI_CURRENT_SCHEMA_XSI,
             Constants.ATTR_TYPE,
             "array item", false);
      QName actualItemType = declItemType != null
                        ? declItemType
                        : arrayItemType;
      Bean itemBean = xjmr.unmarshall(actualEncStyle,
                        actualItemType,
                        tempEl, ctx);
      // get the position in the array
      String pos = DOMUtils.getAttributeNS(tempEl,
                         inScopeEncStyle, "position");
      int right = pos.indexOf(']');
      String substr = pos.substring(1, right);
      int idx = Integer.parseInt(substr);
      Array.set(array, idx, itemBean.value);
      tempEl = DOMUtils.getNextSiblingElement(tempEl);
   }
   return new Bean(array.getClass(  ), array);
}
public static Object getNewArray(String inScopeEncStyle,
               Element arrayEl,
               QName arrayItemType,
               XMLJavaMappingRegistry xjmr)
           throws IllegalArgumentException {
   QName arrayTypeValue =
           SoapEncUtils.getAttributeValue(arrayEl,
           Constants.NS_URI_SOAP_ENC,
           Constants.ATTR_ARRAY_TYPE,
           "array", true);
   String arrayTypeValueNamespaceURI =
           arrayTypeValue.getNamespaceURI(  );
   String arrayTypeValueLocalPart =
           arrayTypeValue.getLocalPart(  );
   int leftBracketIndex =
           arrayTypeValueLocalPart.lastIndexOf('[');
   int rightBracketIndex =
           arrayTypeValueLocalPart.lastIndexOf(']');
   if (leftBracketIndex == -1
     || rightBracketIndex == -1
     || rightBracketIndex < leftBracketIndex) {
       throw new IllegalArgumentException(
               "Malformed arrayTypeValue '" +
                arrayTypeValue + "'.");
   }
   String componentTypeName =
         arrayTypeValueLocalPart.substring(0,
         leftBracketIndex);
   if (componentTypeName.endsWith("]")) {
      throw new IllegalArgumentException(
         "Arrays of arrays are not " +
         "supported '" + arrayTypeValue + "'.");
   }
   arrayItemType.setNamespaceURI(arrayTypeValueNamespaceURI);
   arrayItemType.setLocalPart(componentTypeName);
   int length = DOMUtils.countKids(arrayEl,
       Node.ELEMENT_NODE);
   String lengthStr =
       arrayTypeValueLocalPart.substring(leftBracketIndex + 1,
                                   rightBracketIndex);
   if (lengthStr.length(  ) > 0) {
      if (lengthStr.indexOf(',') != -1) {
         throw new IllegalArgumentException(
              "Multi-dimensional arrays are " +
              "not supported '" + lengthStr + "'.");
      }
      try {
         int explicitLength = Integer.parseInt(lengthStr);
         length = explicitLength;
      }
      catch (NumberFormatException e) {
         throw new IllegalArgumentException(
                "Explicit array length is not a " +
                "valid integer '" + lengthStr + "'.");
      }
   }
   Class componentType = xjmr.queryJavaType(arrayItemType,
                inScopeEncStyle);
   return Array.newInstance(componentType, length);
}
}
```

Let's look at the code for the `marshall( )` method first. This method is called when the array is serialized. The serialization of a sparse array doesn't require that the array elements be treated differently. There are no additional attributes, and the attribute values used are identical to those used for a regularly serialized array. The only change is that each item in the array requires a `position` attribute that indicates where in the array that element is located. The position element needs to be namespace-qualified using `http://schemas.xmlsoap.org/soap/encoding/`. We also want the serializer to skip over any array items that are `null`, instead of including them in the serialization.

Look at the `marshall( )` code where the array length is determined by calling `Array.getLength( )`. The value is returned in the integer variable `length`, which is used in the subsequent `for` loop. Inside this loop, the code works with each item of the array.[2] We get a reference to each element as an `Object` named `value`, and check whether the object is `null`. If it is, we don't care about it and we loop around for the next item. If it's not `null`, we start by determining the runtime class of the `value` object. Next, we want to start writing out the element tag for the array item, but Apache SOAP doesn't provide a mechanism for serializing element attributes. This presents a problem, since we have to include a `position` attribute for every item element of a sparse array. However, we still want to take advantage of all the Apache SOAP code that handles the writing of elements and their values.

Let's take care of the latter part first. An instance of `java.io.StringWriter` called `sw` is created. Normally we'd pass the `sink` to the `xjmr.marshall( )` method, which serializes the item and writes it to the `sink`. Instead of passing the `sink`, we pass the `StringWriter` instance, `sw`. This stores the serialized form of the array item, which we'll use shortly. The fourth parameter passed to `xjmr.marshall( )` is `"item"`, which is the tag name for the element. Since that's the next thing we really want written to the `sink`, we do it ourselves by calling `sink.write` with a parameter of `"<item "`. Here's our chance to stuff an attribute/value pair into the `sink`, because we're positioned exactly where we want to be after writing the item tag. We need to namespace-qualify the attribute, so `nsStatck.getPrefixFromURI( )` is used to find the namespace ID being used for the SOAP encoding namespace. The constant `Constants.NS_URI_SOAP_ENC` represents that namespace, so that is passed as the parameter to `getPrefixFromURI( )`. We start the process of building the entire attribute/value pair for the `position` attribute using the namespace ID that we just retrieved. We know the attribute name is `position`, and that the array item index corresponds to the value of the loop variable `i`. All this is formatted by concatenating these values and passing the result to `sink.write( )`. Since the value of the `position` attribute gets enclosed in quotes, we used escape sequences to get those quotes into the string.

[2] In practice, this is probably not the best way to represent a sparse array in Java, as it still requires you to loop over the entire array. Very large sparse arrays with very few actual entries should be implemented using a more intelligent representation.

Next, we want to get the serialized data from the temporary `StringWriter` into the `sink`. But we don't want the element tag at the beginning because we've already written the tag and its attributes to the `sink`. Since we used the same tag name, `item`, we can skip the first six characters of the data in the `StringWriter` instance and grab the rest. (The six characters that we want to skip are the opening <, plus `item`, plus the trailing space that follows.) So we call `sw.toString( ).substring(6)` and pass that to `sink.write( )`. We've managed to get the attribute into the item element and still take advantage of the Apache SOAP API to do most of the work. The remainder of the code in `marshall( )` adds a line break and the closing tag to the `sink` to finish off the serialization of the array element.

Next let's look at the `unmarshall( )` method used to deserialize a sparse array. In this case, we don't have much work to do. Almost all the code centers around determining the array size and type, and allocating an appropriate array instance. We just want to determine the value of the `position` attribute so that we can put the deserialized item instance into the proper position of the array. We can get the string value of the attribute by calling `DOMUtils.getAttributeNS` . We pass the method the element of the XML document, the encoding style that is currently in-scope, and the name of the attribute. The element is in the variable `tempEl`, which was handled earlier in the method. The encoding style is found in the `inScopeEncStyle` parameter that was passed to the `unmarshall( )` method. And, of course, we know that the attribute name is `position`. The value of the attribute will be an integer enclosed in square brackets. So we pull out the substring between the brackets and pass it to `Integer.parseInt( )` to get an `int` value that represents the array position. This value is passed as the second parameter to the `Array.set( )` method, which places the item into the array at the specified position. The rest was already written for us.

Now that we've got a sparse array serializer, we need a way to use it. After all, Apache SOAP, by default, wants to use the `ArraySerializer` class to handle the serialization and deserialization of arrays. So we need to find a way to override that behavior. My first inclination would be to modify the `org.apache.soap.encoding.SOAPMappingRegistry` class provided as part of Apache SOAP to force it to use the `SparseArraySerializer` instead of the `ArraySerializer`. However, one problem with this approach is that it renders invalid any arrays serialized without `position` parameters. That doesn't sound good. We could have made the `SparseArraySerializer` smart enough to handle both possibilities, but that would have meant a highly complex serializer with a lot more code. If you'd like to take that on, I bet the Apache folks would be happy to consider including it. A simpler approach would be to map the sparse array serializer in the mapping section of those services that use sparse arrays. This means that we don't have to modify the Apache SOAP source code, which could become a real pain if changes in future versions of the Apache code are inconsistent with our modified version. So we'll use the service-specific approach.

Let's add a method called `echoStocks` to the `urn:DataFeedService` service. This method takes a string array as a parameter and uses that same array as the return value. Here's the new version of the `DataFeedService` class with the Java implementation, `echoStocks( )`, added:

```
package javasoap.book.ch6;
public class DataFeedService {
   public DataFeedService( ) {
   }
   public String[] echoStocks(String[] stocks) {
      return stocks;
   }
   public DataFeedMessage sendMessage(DataFeedMessage msg) {
      // we could pull the formatted message here
      // by calling msg.toString(  );
      String stock = msg.getFieldValue("SYMBOL");
      String request = msg.getFieldValue("REQUEST");
      DataFeedMessage res = new DataFeedMessage(  );
      res.addField("SYMBOL", stock);
      if (request.equals("PRICE"))
         res.addField("PRICE", "123.5");
      return res;
   }
}
```

We need to add a mapping entry to the deployment descriptor so that the sparse array serializer is used for arrays. The type that we want to map is `Array`, namespace- qualified by `http://schemas.xmlsoap.org/soap/encoding/`. That's the standard SOAP array using standard SOAP encoding. This mapping entry overrides the default classes for serializing and deserializing arrays. We already know that the `SparseArraySerializer` class will be used as the value for both the `java2XMLClassName` and the `xml2JavaClassName` attributes. That tells the system to use our new serializer for arrays. Now we need to specify the Java class that's used to instantiate a Java array. That class comes from the Java reflection API, and is named `java.lang.reflect.Array`. We use this value for the `javaType` attribute of the mapping entry. The `echoStocks` method name is added to the `methods` attribute of the `service` tag.

We still need to work around a problem, however. The Apache SOAP Java code doesn't expect you to override the standard serializers, and makes some assumptions about the serializers and mapping registry. Even though we've specified the `SparseArraySerializer` for the `java2xmlClassName` in the mapping entry, it doesn't get used when our service attempts to return an array from `echoStocks`. The workaround is to write our own mapping registry, which is easier than it sounds. We'll do this shortly; the new registry will be named `javasoap.book.ch6.BetterSOAPMappingRegistry`. We need to specify this registry in the deployment descriptor by including the `defaultRegistryClass` attribute in the `isd:mappings` element, using the class name of our new registry as the value. Here's the modified deployment descriptor:

```
<isd:service
    xmlns:isd="http://xml.apache.org/xml-soap/deployment"
    id="urn:DataFeedService">
  <isd:provider
     type="java"
     scope="Application"
     methods="sendMessage echoStocks">
    <isd:java
       class="javasoap.book.ch6.DataFeedService"
       static="false"/>
  </isd:provider>

  <isd:faultListener>org.apache.soap.server.DOMFaultListener
  </isd:faultListener>
  <isd:mappings
     defaultRegistryClass="javasoap.book.ch6.BetterSOAPMappingRegistry" >
     <isd:map
       encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
       xmlns:x="http://schemas.xmlsoap.org/soap/encoding/" qname="x:Array"
       javaType="java.lang.reflect.Array"
       java2XMLClassName="javasoap.book.ch6.SparseArraySerializer"
       xml2JavaClassName="javasoap.book.ch6.SparseArraySerializer"/>
     <isd:map
       encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
       xmlns:x="urn:DataFeedService" qname="x:DataFeedMessage"
       javaType="javasoap.book.ch6.DataFeedMessage"
       java2XMLClassName="javasoap.book.ch6.DataFeedMessageSerializer"
       xml2JavaClassName="javasoap.book.ch6.DataFeedMessageSerializer"/>
  </isd:mappings>
</isd:service>
```

All our new mapping registry needs to do is return an instance of the `SparseArraySerializer` instead of `ArraySerializer` when it gets queried for an array serializer. The `BetterSOAPMappingRegistry` extends the `SOAPMappingRegistry` class provided by Apache SOAP. To solve the problem, we override the `querySerializer` method so that it returns the `SparseArraySerializer` when needed. The base class is doing most of the work, but if it wants to return an `ArraySerializer`, we return a `SparseArraySerializer` instead. Here's the new registry class:

```
package javasoap.book.ch6;
import org.apache.soap.util.xml.*;
import org.apache.soap.*;
import org.apache.soap.encoding.soapenc.*;
public class BetterSOAPMappingRegistry
       extends org.apache.soap.encoding.SOAPMappingRegistry {
   SparseArraySerializer sparseSer = new SparseArraySerializer(  );
   public BetterSOAPMappingRegistry(  ) {
      super(  );
   }
   public Serializer querySerializer(Class javaType,
                  String encodingStyleURI)
        throws IllegalArgumentException {

      Serializer s;
      try {
         s = super.querySerializer(javaType, encodingStyleURI);
         if (s instanceof ArraySerializer)
           return sparseSer;
      }
      catch (IllegalArgumentException e) {
         if (javaType != null
          && encodingStyleURI != null
          && encodingStyleURI.equals(Constants.NS_URI_SOAP_ENC)) {
            if (javaType.isArray(  )) {
              return sparseSer;
            }
         }
         throw e;
      }
      return s;
   }
}
```

Why do we need a mapping entry in the deployment descriptor if we're using the `defaultRegistryClass` attribute of the `isd:service` element? Well, because the attribute by itself doesn't seem to work either, but the combination of the attribute and the mapping entry does. The workaround is fairly simple, and saves you from going deeply into the Apache source code. Go ahead and deploy the service.

Now, let's see what has to be done in a client application. We actually need to do the same thing, but it's accomplished in a different way. We specify the mapping, and use the `BetterSOAPMappingRegistry` class instead of the `SOAPMappingRegistry` class. Here's the client code:

```
package javasoap.book.ch6;
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.encoding.*;
import org.apache.soap.util.xml.*;
public class DataFeedClient2 {
   public static void main(String[] args)
      throws Exception {

      URL url = new URL(
         "http://georgetown:8080/soap/servlet/rpcrouter");

      Call call = new Call(  );

      BetterSOAPMappingRegistry smr =
                  new BetterSOAPMappingRegistry(  );
      call.setTargetObjectURI("urn:DataFeedService");
      call.setMethodName("echoStocks");
      call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
      call.setSOAPMappingRegistry(smr);
      SparseArraySerializer arraySer =
                  new SparseArraySerializer(  );
      // Map the array
      smr.mapTypes(Constants.NS_URI_SOAP_ENC,
               new QName(Constants.NS_URI_SOAP_ENC, "Array"),
               java.lang.reflect.Array.class, arraySer,
               arraySer);
      String[] stocks = new String[10];
      stocks[2] = "XYZ";
      stocks[5] = "ABC";
      Vector params = new Vector(  );
      params.addElement(new Parameter("msg", String[].class,
                           stocks, null));
      call.setParams(params);

      try {
         Response resp = call.invoke(url, "");
         Parameter ret = resp.getReturnValue(  );
         String[] value = (String[])ret.getValue(  );
         int cnt = value.length;
         for (int i = 0; i < cnt; i++) {
            if (value[i] != null) {
               System.out.println("Item " + i + ": " + value[i]);
            }
         }
      }
      catch (SOAPException e) {
         System.err.println("Caught SOAPException (" +
                        e.getFaultCode( ) + "): " +
                        e.getMessage( ));
      }
   }
}
```

The `smr` variable is now an instance of `BetterSOAPMappingRegistry`, but from that point on we use it just as in the previous examples. We create an instance of `SparseArraySerializer` to use for the mapping process; then we call `smr.mapTypes( )`, giving it the previously instantiated `SparseArraySerializer` as the object to use for both serializing and deserializing, and `java.lang.reflect.Array.class` as the class to use for the object that implements the array. Take note of the parameters used for the `Qname` constructor. The first parameter is the standard SOAP encoding namespace, and the second is `"Array"`. So this should override the default mapping.

Now we create an empty string array of size 10, and assign values to positions 2 and 5. The rest of the items in the array are null. Don't assign empty strings here, because an empty string is not the same as a null value. (Although there's actually no harm in trying empty strings; you will get a serialized element for all 10 items in the array.) The `echoStocks` service method is invoked the same way as we've done before. The return value is cast to a `String[]` so that we can iterate through it looking for the values. If all is correct, we should find objects only at positions 2 and 5, since the server echoed back the same array we sent to it. Running this application results in the following output:

```
Item 2: XYZ
Item 5: ABC
```

And here's the SOAP envelope for this example. If nothing else, this shows that only the non-null elements were actually serialized.

```
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Body>
 <ns1:echoStocks
   xmlns:ns1="urn:DataFeedService"
   SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
     <msg
       xmlns:ns2="http://schemas.xmlsoap.org/soap/encoding/"
       xsi:type="ns2:Array" ns2:arrayType="xsd:string[10]">
         <item ns2:position="[2]" xsi:type="xsd:string">XYZ</item>
         <item ns2:position="[5]" xsi:type="xsd:string">ABC</item>
     </msg>
 </ns1:echoStocks>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The `msg` element is typed as an array of 10 strings, no different from any other array. Each `item` element is explicitly typed as an `xsd:string`, and includes a `position` attribute that indicates its position in the array. That's exactly what we were shooting for.

6.1.2 Serializing Collections

Arrays aren't the only collection types commonly used in Java programming. It would also be nice to be able to serialize hash tables, linked lists, and other collection types. GLUE provides a custom implementation for a hash table that's worth looking at. This example uses just one of the many mechanisms that GLUE provides for custom serialization. It requires you to write the custom type schema on the fly, which of course means you'll need to understand schemas. That's not the subject of this book, but it's still worth looking at that part of the code.

To write a custom serializer in GLUE, start with a Java class that extends `electric.xml.io.Type`, the base class for SOAP types in GLUE. All we need to do is override the methods `writeSchema( )` , `writeObject( )`, and `readObject( )`. Here is GLUE's implementation of the `HashtableType` class:

```
// copyright 2001 by The Mind Electric
package electric.xml.io.collections;
import java.io.*;
import java.util.*;
import electric.xml.*;
import electric.xml.io.*;
import electric.util.Value;
public class HashtableType extends Type
  {
  /**
   * @param schema
   */
  public void writeSchema( Element schema )
    {
    Element complexType = schema.addElement( "complexType" );
    complexType.setAttribute( "name", "Map" );
    Element sequence = complexType.addElement( "sequence" );
    Element element = sequence.addElement( "element" );
    element.setAttribute( "name", "item" );
    element.setAttribute( "minOccurs", "0" );
    element.setAttribute( "maxOccurs", "unbounded" );
    Element subType = element.addElement( "complexType" );
    Element subSequence = subType.addElement( "sequence" );
    Element key = subSequence.addElement( "element" );
    key.setAttribute( "name", "key" );
    key.setAttribute( "type", getName( key, Object.class ) );
    Element value = subSequence.addElement( "element" );
    value.setAttribute( "name", "value" );
    value.setAttribute( "type", getName( key, Object.class ) );
    }
  /**
   * @param writer
   * @param object
   * @throws IOException
   */
  public void writeObject( IWriter writer, Object object )
    throws IOException
    {
    writer.writeType( this );
    Hashtable hashtable = (Hashtable) object;
    for( Enumeration enum = hashtable.keys(  );
                              enum.hasMoreElements(  ); )
      {
      Object key = enum.nextElement(  );
      Object value = hashtable.get( key );
      IWriter itemWriter = writer.writeElement( "item" );
      itemWriter.writeObject( "key", key );
      itemWriter.writeObject( "value", value );
      }
    }
  /**
   * @param reader
   * @param value
   * @throws IOException
   */
  public void readObject( IReader reader, Value value )
    throws IOException
    {
    Hashtable hashtable = new Hashtable(  );
    value.setObject( hashtable );
    IReader[] readers = reader.getReaders( "item" );
    for( int i = 0; i < readers.length; i++ )
      {
      IReader itemReader = readers[ i ];
      Object key = itemReader.readObject( "key" );
      Object theValue = itemReader.readObject( "value" );
      hashtable.put( key, theValue );
      }
    }
  }
```

GLUE provides built-in support for this class, so we won't have to register it when we write the service and client application code. If you want to develop your own custom type serializers in GLUE, you must use one of the `mapClass(  )` methods from the `electric.xml.io.Mappings` class. These methods tell the GLUE system how to map your custom types to serializers.

The code in the `writeSchema(  )` method writes a schema definition for the hash table. Instead of looking at each class and method used to accomplish this, let's look at the resulting schema:

```
<complexType name="Map">
  <sequence>
    <element name="item" minOccurs="0" maxOccurs="unbounded">
      <complexType>
        <sequence>
          <element name="key" type="KEYTYPE"/>
          <element name="value" type="VALUETYPE"/>
        </sequence>
      </complexType>
    </element>
  </sequence>
</complexType>
```

This is a schema definition for a hash table, otherwise known as a map. Remember that each entry in a map contains a key/value pair. It's defined as a `complexType` with the name `Map`, containing a sequence of 0 or more elements. Each element is, in turn, also a complex type containing a sequence of two elements, `key` and `value`. These two elements are typed using the appropriate type names for the `key` and `value` elements; I used `KEYTYPE` and `VALUETYPE` as placeholders. In the code, those types are determined from the class of the corresponding

Java object instance. We'll see this shortly when we run an example using this custom serializer.

The `writeObject( )` method is where the contents of the custom type are serialized. The first parameter of the method is an instance of the `electric.xml.io.IWriter` interface. We use this interface to write the contents of the object being serialized. The second parameter is the Java object to be serialized. The method first writes the object's type to the stream, passing `this` to the `writeType( )` method of the `writer`. This identifies the type of the object being serialized as a `Map`. Then the object is cast to a `Hashtable` instance so that we can access the key/value pairs. All we need to do is iterate over the elements of the table, using the `Enumeration` returned from the `hashtable.keys( )` method. For each element, we retrieve the `Object` instances for the key and the value. To write each item, we call `writer.writeElement( )`, which returns a new instance of `IWriter` called `itemWriter`. This instance is used for writing out the key and value data for that item using `itemWriter.writeObject( )`, which is given the element name and the associated `Object` instance as parameters.

`readObject( )` performs the deserialization. Its first parameter is an instance of the `electric.xml.io.IReader` interface, which we use to read the contents to be deserialized. The other parameter is an instance of `electric.util.Value`, which contains a `java.lang.Object` instance and its associated type (an instance of `java.lang.Class`). The first step is to create a new instance of `java.util.Hashtable` and pass it as a parameter to `value.setObject( )`. Next we call `reader.getReaders( )`, which returns an array of `IReader` instances, one for each key/value pair in the map. For each `IReader`, we call `getObject( )` to retrieve the key and value objects for the element. These two objects are now added to the hash table using the `hashtable.put( )` method.

That's all there is to it. So we'll now build a service and a client application that use hash tables to see how the custom serializer gets registered. Let's create a new service called `urn:AnotherFeedService`. This one, like the data feed service from before, contains a method called `sendMessage( )`. But this time the method takes a hash table as the parameter, where the elements of the hash table contain the field names and their associated values. This means that the new service must format the data as specified at the beginning of this chapter. We won't bother with that, since it doesn't have any real impact on what we're doing. We'll just concentrate on the custom serialization of hash tables. The `javasoap.book.ch6.AnotherFeedService` class implements the service.

```
package javasoap.book.ch6;
import java.util.Hashtable;
public class AnotherFeedService {
   public AnotherFeedService(  ) {
   }
   public Hashtable sendMessage(Hashtable msg) {
      String stock = (String)msg.get("SYMBOL");
      String request = (String)msg.get("REQUEST");
      Hashtable res = new Hashtable(  );
      res.put("SYMBOL", stock);
      if (request.equals("PRICE"))
         res.put("PRICE", "123.5");
      return res;
   }
}
```

The `sendMessage( )` method performs essentially the same function as it did in the earlier `DataFeedService` class. The difference is that the messages are passed using hash tables instead of instances of `DataFeedMessage`. To deploy the service, we use the class `FeedServiceApp`:

```
package javasoap.book.ch6;
import electric.util.Context;
import electric.registry.Registry;
import electric.server.http.HTTP;
import java.util.Hashtable;
public class FeedServiceApp {

   public static void main( String[] args )
                  throws Exception {
      String ns = "urn:AnotherFeedService";
      HTTP.startup("http://georgetown:8004/glue");
      Context context = new Context( );
      context.addProperty("activation", "application");
      context.addProperty("namespace", ns);
      Registry.publish(ns,
         javasoap.book.ch6.AnotherFeedService.class, context );
   }
}
```

Let's write the interface for binding to the service by hand, since we're taking advantage of a custom type that's already built in to GLUE. Here's the `javasoap.book.ch6.IAnotherDataFeed` interface. It contains the `sendMessage( )` method that we defined for the service, which takes a `Hashtable` as a parameter and also returns one.

```
package javasoap.book.ch6;
public interface IAnotherFeedService {
  java.util.Hashtable sendMessage(java.util.Hashtable msg);
}
```

`FeedServiceClient` is an application that binds to the `urn:AnotherFeedService` service and passes a hash table to the `sendMessage( )` method.

```
package javasoap.book.ch6;

import electric.registry.RegistryException;
import electric.registry.Registry;
import java.util.*;

public class FeedServiceClient {
   public static void main(String[] args) throws Exception
   {
      try {

         IAnotherFeedService srv = (IAnotherFeedService)Registry.bind(
          "http://georgetown:8004/glue/urn:AnotherFeedService.wsdl",
           IAnotherFeedService.class);

         Hashtable msg = new Hashtable( );
         msg.put("SYMBOL", "XYZ");
         msg.put("REQUEST", "PRICE");
         Hashtable result = srv.sendMessage(msg);

         for (Enumeration e = result.keys(); e.hasMoreElements( ); ) {
            String key = (String)e.nextElement( );
            String value = (String)result.get(key);
            System.out.println(key + ": " + value);
         }
      }
      catch (RegistryException e) {
         System.out.println(e);
      }
   }
}
```

The returned hash table is enumerated, and the key/value pairs are printed for display. The result is:

```
SYMBOL: XYZ
PRICE: 123.5
```

Let's take a look at the SOAP envelope that was sent to the service when `sendMessage( )` was invoked. This gives us a chance to see the serialized hash table.

```
<soap:Envelope
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:xsd='http://www.w3.org/2001/XMLSchema'
  xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'
  xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
  soap:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
    <soap:Body>
      <n:sendMessageResponse xmlns:n='urn:AnotherFeedService'>
        <Result href='#id0'/>
      </n:sendMessageResponse>
      <id0 id='id0' soapenc:root='0'
            xmlns:ns2='http://xml.apache.org/xml-soap'
            xsi:type='ns2:Map'>
        <item>
          <key xsi:type='xsd:string'>PRICE</key>
          <value xsi:type='xsd:string'>123.5</value>
        </item>
        <item>
          <key xsi:type='xsd:string'>SYMBOL</key>
          <value xsi:type='xsd:string'>XYZ</value>
        </item>
      </id0>
    </soap:Body>
</soap:Envelope>
```

Just as we've seen in all the other GLUE examples using complex types, the actual data is serialized separately, and the parameter references that data. The hash table we sent can be found in the `id0` element. It is typed as a `Map`, qualified by the `http://xml.apache.org/xml-soap` namespace. This namespace is used because the Apache group actually created the schema for encoding a map, or hash table, type. The GLUE implementation is consistent with the Apache work in this area. Each `item` child element contains `key` and `value` elements that in turn contain the key/value pair for the map entry. In this case, all the keys and values are strings because that's the type we used in the example.

In this chapter, we looked at how to create serializers for our own custom types. Using these techniques combined with the built-in support for many types, there is virtually no limit to the data that can be transmitted using SOAP. In some cases you may find that you're designing your classes with SOAP in mind, or least with the capabilities and restrictions of your chosen SOAP implementation in mind. Whether or not designing around the limitations of your SOAP implementation makes sense for your project, custom serialization opens up a lot of possibilities when you need to go beyond the basics.